DATA WAREHOUSE ARCHITECTURE FOR DSS APPLICATIONS

V. J. Anand and Himawan Gunadhi†

Department of Decision Sciences
National University of Singapore
10 Kent Ridge Crescent
Singapore 119260

ABSTRACT

A *data warehouse* facilitates the integration of disparate operational databases in an enterprise into a single store. The warehouse then provides *knowledge workers* with easy access to historical, summarized and other forms of aggregated data. A major flaw in present warehouse architectures is the de-coupling of the warehouse database from its underlying operational databases. This creates two problems: the difficulty in updating the warehouse and the inability to provide users with a drill-down feature from warehouse to current data. We propose an architecture that is based on a *federated database system* extended to incorporate a materialized warehouse. A federated database provides the basic mechanism to tightly couple heterogeneous databases, which in this case are the operational and warehouse databases. The warehouse database is a special component in the federation, made up of historical data, external data and *materialized views* of the operational data. This approach enables users to access historical and current data when required, and provides a method of maintaining the warehouse as an integrated view of the underlying operational data sources.

INTRODUCTION

Database management systems have helped enterprises to efficiently collect and manage data, but have fallen short in providing an environment where this data can be used to gain a better business insight and making quality decisions. The problem arises from the manner in which databases have been built and deployed – primarily for storing transaction data generated by the day-to-day activities of an organization. In addition, large corporations have built disparate and heterogeneous database systems, each fine tuned for a specific function. Consequently, several requirements of knowledge workers have not been satisfied: 1) Access to historical data that cut across functional boundaries and involve complex aggregates; 2) Quick turnaround between the request and availability of information; and 3) The ability of operational systems to simultaneously cater to the needs of both operational and decision support applications.

Data warehousing [Inmon (1992); Inmon & Hackathorn (1994); Kelly (1994); Gupta et. al (1995); Hackathorn (1995)] provide a framework for integrating data from multiple databases in a logical and unified fashion. It provides users with easy access to enterprise data in a consistent manner and a method for storing historical, summarized and other data aggregations. Separating the data required for decision making activities from those needed for operations provide the following advantages: 1) Integration of data from multiple sources in a consistent manner; 2) A single point access for enterprise-wide data; 3) Enhanced efficiency in terms of availability and response time for user queries; and 4) Real-time analysis of data for decision-making, i.e., On-Line Analytical Processing (OLAP).

To provide users with consistent and current information, a data warehouse has to be updated constantly. Present methods [Inmon & Hackathorn (1994)] use programs that periodically extract data from different sources. Since these are batch processes, the data warehouse is updated only periodically, say once a day. Thus, the warehouse is as current as the last extraction cycle. Also, these programs have to be altered against any schema change either in the operational databases or data warehouse, since these affect the mapping information. While the de-coupling of decision-support and transaction-oriented databases have resulted in enhanced response time for complex queries, it has also created an island between them. There are situations when a data warehouse user requires access to the most current data, but there is no easy access path from the warehouse to the sources residing in the operational databases.

Research has recently been carried out in the area of warehouse maintenance as a materialized view of the underlying data sources [Zhou et. al (1995)]. This enables existing view maintenance methods [Hanson (1987)] to be adapted. But, a warehouse is not created simply by joins or projections among tables from a single data source – some data are derived by complex aggregations that span multiple databases. Moreover, many of the sources are not relational databases but legacy-based systems, e.g., hierarchical and network databases, which may not support views. The Stanford WHIPS project [Hammer et. al (1995)] models a warehouse as a materialized view that can be incrementally maintained via an integrator and monitors. Whenever a change is detected in the data source, the corresponding monitor sends a message to the integrator to update the warehouse. The integrator may subsequently require additional data either from the same or additional sources. The drawback of this model is that it becomes difficult to maintain the integrator against changes in the schema

† Author's present address: 1691 Ruther Place Ct., San Jose, CA 95121, USA. Internet: h.gunadhi@usa.net.

at the data source or warehouse. Moreover, the integrator has to respond to a change detected by a monitor by sending appropriate queries to the data sources, resulting in increased maintenance costs.
Research in heterogeneous databases [Sheth & Larson (1990)] have provided methods to integrate disparate databases into a single unifying architecture — the federated database model. We propose to extend the federated architecture to include a data warehouse modeled as a materialized view of the underlying federated schema. In addition, it adapts a *deferred* view maintenance approach, rather than *immediate* approach adopted by WHIPS. This approach is preferred, because a great deal of decision-making may not require current data, but for those that require them, the model provides a mechanism to obtain them without adding too much overhead.
This approach provides the following advantages:

- A unified architecture that ties the data warehouse to multiple heterogeneous databases.
- Provides a method of maintaining the data warehouse as an integrated materialized view of the underlying data sources.
- Provides flexible access to current data residing in the data sources.
- Ease of maintenance against any change to the schema in the data source or warehouse.

The rest of the paper is organized as follows: Section 2 gives an overview of related research that forms the basis for this paper. Section 3 introduces the integrated data warehouse model and the concept of schemes within this framework. Section 4 describes how view management is handled by the architecture, while Section 5 concludes by outlining future research.

RELATED RESEARCH

In this section we give an overview of research in federated database systems, data warehousing and view maintenance which are closely related to the work we describe in this paper.

Federated Database Systems

Federated database systems provide methods of integrating heterogeneous databases into a unified system, thereby increasing the accessibility. In other words, users have to interact with a single system rather than individual databases. Federated database systems provide: 1) methods for data integration — reconciling, cleaning and removing inconsistencies between the various schemes to generate a *federated schema* [Sheth & Larson (1990)], and 2) transparent access to users — the system takes care of coordinating in splitting the query and subsequently combining the results. Federated databases are implemented as either loosely or tightly coupled architectures [Sheth & Larson (1990)] depending on the level of autonomy. Though they provide ease of access they are limited in as much as: 1) data that can be seen by all users is dependent on individual databases, 2) federated schema is non-materialized, which means that queries have to be evaluated in the individual database, resulting in slower response time, and 3) data from external sources are not integrated within the federated schema.

Data Warehouse

Data warehouses [Inmon (1992)] are large repositories for analytical data. They are primarily designed to improve query response time and provide data in a form required by the knowledge workers. They offload the query overhead from operational databases into a separate specialized database providing improved efficiency. In addition, they also provide a means of: 1) integrating data; both external and enterprise in a consistent manner, and 2) storing different forms of data, e.g., data cubes [Weldon (1995)], aggregates of data, etc. Presently these systems are deployed separately, without any link to the underlying databases and data is periodically pushed or pulled into the warehouse. But for such queries that require access to the operational data from the warehouse, it does not provide a method.

View Maintenance

Materialized views [Gupta & Mumick (1995)] provide a means of caching the result of a query and therefore, providing increased query response time. This technique has been widely used in distributed databases [Segev & Park (1989)], where base relations and views are physically separate. Materializing the view requires it to be constantly updated to reflect the present state of the base relations. A trivial and most often used method is to re-compute the view each time the base relation undergoes change. The problem with this approach is that base

relations that change often can trigger excessive re-computation of the view, although it is efficient for certain operations as deleting the whole base relation. Alternatively, only changes that occur on the base tables are propagated incrementally. Based on *when* the view is updated, the algorithms [Hanson (1987)] can be classified into: 1) immediate — which propagates the change within the transaction that affects the base relation, and 2) deferred — which propagate the change after a time lag.

Immediate propagation [Blakely et. al (1986); Shmueli (1984); Zhuge et. al (1995)] methods have been well researched and a wide range of algorithms that implement such a technique is reported in the literature. In a number of applications, immediate view maintenance is simply not possible. Because applications cannot tolerate any slow down, systems would like to keep the transaction overhead to the minimal. Moreover, in a distributed scenario, the de-coupling of the base relation from the view can result in update anomalies [Zhuge et. al (1995)] and high network cost.

Deferred view maintenance [Agarwal & Dewitt (1983); Hanson (1987); Griffin & Libkin (1995)] has however not been studied as extensively as the immediate case. They propagate the change periodically or on-demand when certain conditions arise. Therefore, the materialized view is inconsistent with the base relations for a period of time, which is tolerable for a number of applications. Moreover, they do not add any overhead to the transaction as they require only to insert an entry into a log file. In addition, efficiency is gained by batching several updates together and replacing several small queries and updates into one.

THE INTEGRATED DATA WAREHOUSE MODEL (IDWM)

Integrated data warehouse model integrates a data warehouse into a tightly coupled federated database architecture (Figure 1). This provides a three tier architecture: operational data sources and data warehouse forms the bottom and top layers respectively, while the interface layer provides the mapping between the two. When access to the current data is required it is retrieved transparently from the operational data sources by accessing the interface layer. In the model, a *component database* (CDB) refers to an operational data source. Data that is of interest is first represented uniformly through the *export schema*, which defines data that is exported from the operational sources.

After reconciling, integrating and removing semantic heterogeneity between various export schemes, these are then integrated along with the external data into an *integrated data schema* (IDS) depicted near the top of figure 1. Here, we make a distinction between IDS and the federated database schema as defined in the literature. The former refers to a schema that integrates all data from the participating operational sources, while the latter may not. In addition, we distinguish export schema from component schema only in the type of data model that is employed in defining them, since we assume that all data from the component database is for export. For example, component schema can be defined as hierarchical and the export schema can be defined as relational.

Between the operational sources and the data warehouse lies the *Global View Management System* (GVMS), and the *Local View Manager* (LVMs). An LVM interfaces the operational source into the framework, and provides utilities to generate differential files and specific mapping utility between the export and local schema. In addition, it periodically, sends updates from the operational source to the warehouse.

GVMS services request for data from the warehouse and coordinates LVMs during periodic updates. When updates are received at the warehouse, GVMS has to propagate the change to the materialized export views and the warehouse views. Although, information flow appears to be exclusively "upwards" in direction, there are situations when GVMS requests for data from one or more LVMs.

The model does not make any assumption on the type of operational sources: they can be any database employing any data model e.g., hierarchical, network or object-oriented. Also, they may or may not have a view management facility. Moreover, the model views the warehouse not just as a source of data for decision-making purposes, but also as a repository of reconciled and integrated enterprise data. Thus, there is a natural evolution of a federated database into a data warehouse.

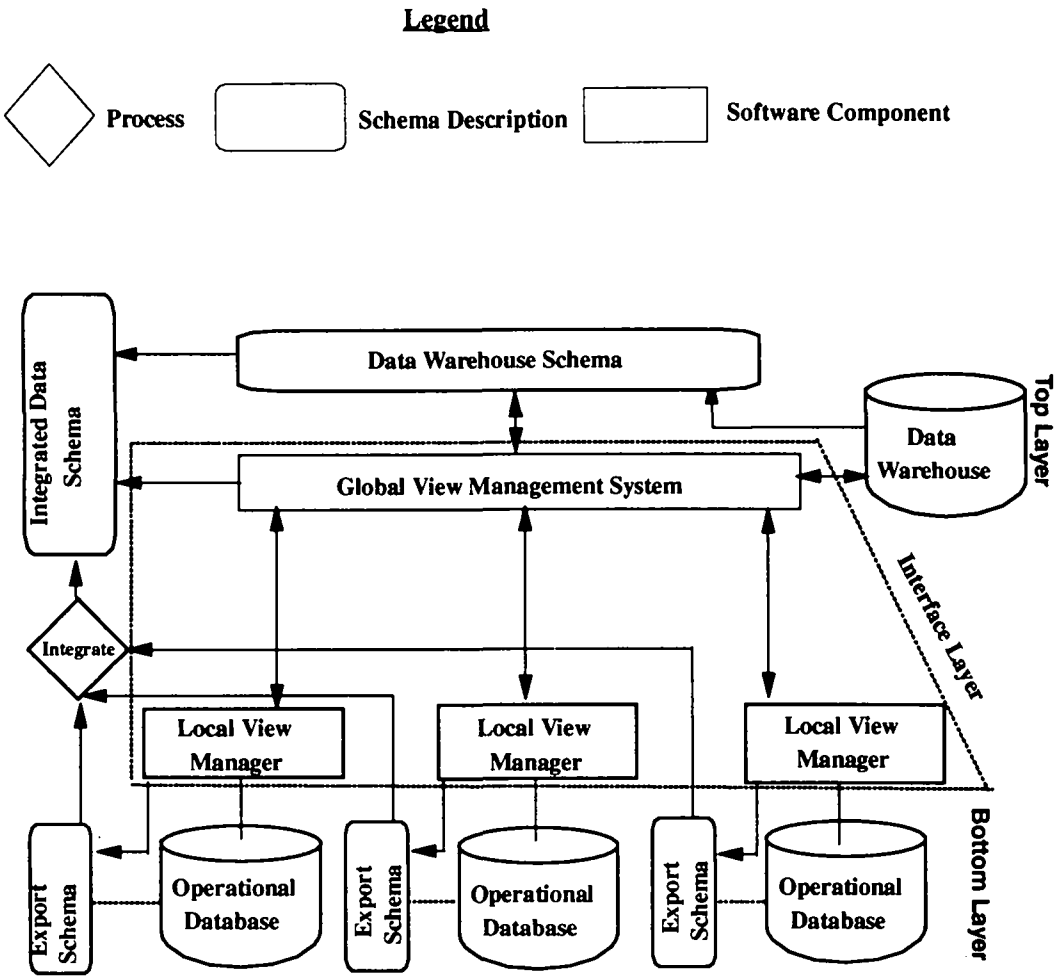

Figure 1. The Integrated Data Warehouse Model

Export Schema

As different data models may be employed by the underlying component databases, it becomes imperative to map them to a common data model. The export schema provides a common representation to which each component schema is mapped. For example, one can represent an individual component schema as a relational export schema, with appropriate mappings between them. This helps in localizing the schema changes, and makes the local database administrator responsible for keeping the export schema consistent. In addition, we consider the export schema not just as a mapped representation of the underlying tables, but as meaningful views. This helps in resolving what needs to be exported from a component database. Unlike a typical federated database system where the local administrators are responsible for exporting data, here the responsible administrator or users can determine that a certain data item is to be obtained from a particular database. As an illustration, there may be two databases maintaining customer names, and both can be exported; but it is decided that only one will export it. Thus, the data item gets included in one, and excluded from the second export schema.

Integrated Data Schema

This schema is generated by combining and reconciling the export schemes from the component databases. It provides a consistent and reconciled view of all the data existing in the enterprise. Also, it is a means of integrating unstructured and external data within the organization. Generally, the IDS follows the same data model as the export schema, but may differ. We consider the former, as this removes an additional mapping between two different data models. In addition, IDS forms the basis from which the schema for the warehouse is extracted. Changes to the warehouse are localized since IDS provides the interface between the warehouse and export scheme of the component database.

Thus, IDS maintains:

- Information regarding the source from where a certain data is obtained.
- Mapping information to the export schemes: this refers to any transformation that is required between the two representations.
- Mapping information to the warehouse schema: this refers to any transformation that is required between the two representations.

Data Warehouse Schema

The Data warehouse schema consists of aggregates, summaries and other data groupings based on different dimensions. Time, regions and product lines are examples of common dimensions. IDS provides the data warehouse with an enterprise data schema from which one can generate such views. Initially, the warehouse schema is small and grows as users become aware of the advantages of using a warehouse. The warehouse may not have all the data to respond to user queries at first; data may have to be obtained from the CDBs transparently, and subsequently integrated into the warehouse schema. This allows a natural evolution of the warehouse schema where usage determines what has to be stored.

We formally define a warehouse as a collection of materialized export and warehouse schemes. Materialized export views are stored over time for access to historical operational data. At any given time, an instance of a warehouse view will correspond to a set of materialized export views and external data mapped through the integrated data schema. Any schema change at the warehouse will not have retrospective effect on its earlier definition. Hence, the affected view will be considered as a new inclusion in the warehouse schema. This allows users to access older versions of the schema as it is still maintained in the warehouse. But, previous versions of the same schema is not materialized in subsequent refreshes.

Let $\{v_1, v_2, \ldots, v_m\}$ be the set of views exported from a single component database. Let $\{g_1, g_1, \ldots, g_m\}$ be the set of transformation functions between the local schema and the export views; g_i is defined as selection, projection, join or combination thereof, over the local schema. Let $\{d_1, \ldots, d_w\}$ be the views defined at the data warehouse. Let $\{f_1, f_2, \ldots, f_w\}$ define the set of transformation functions between the exported views and the views in the warehouse; f_j is defined as selection, projection, join or a combination thereof, over the exported views. Let $\{t_1, t_2, \ldots, t_p\}$ define time intervals when the warehouse is refreshed.

The set of export views from a single component database at the warehouse is defined as:

$$\bigtimes_{i=1}^{m} v_i$$

Export views materialized at the warehouse is given by:

$$\bigtimes_{i=1}^{m} \bigtimes_{t=1}^{p} v_i(t)$$

where v_i refers to intention and $v_i(t)$ to extension.

Views in the warehouse schema defined over export views is given by:

$$\bigtimes_{j=1}^{w} f_j\left(\bigtimes_{i=1}^{m} v_i\right)$$

The materialized warehouse schema can then be defined as:

$$\bigtimes_{j=1}^{w} f_j\left(\bigtimes_{i=1}^{m} v_i\right) + (external\ data)$$

We illustrate the above through an example: Let $r_1(A_{11}, A_{12}, \ldots, A_{1h})$ and $r_2(A_{21}, A_{22}, \ldots, A_{2l})$ be two relations in a component database. Let $v_1 = \sigma_{P1}(r_1)$ and $v_2 = \pi_{(A21, A22, \ldots, A2q)}(r_2)$ be the export views defined over relations r_1 and r_2 respectively. Let $d_k = \sigma_{P1}(v_1) \times_{P2}(\sigma_{P3}(v_2))$ be a view defined in the warehouse over the exported views, where P_1, P_2 and P_3 are condition predicates. Then it follows that

$$\bigtimes_{i=1}^{m}(v_i) = \{v_1, v_2\}$$

$$d_k = \bigtimes_{j=1}^{w} f_j\left(\bigtimes_{i=1}^{m} v_i\right) = \{\sigma_{P1}(v_1) \times_{P2}(\sigma_{P3}(v_2))\}$$

The above definitions and example illustrate the case for a single component database. For multiple component databases, we derive the following equations:

Views in the warehouse schema defined over export views is given by:

$\bigcup_{j=1}^{w} f_j\left(\bigcup_{k=1}^{c}\bigcup_{i=1}^{n} v_{ki}\right)$, where $k = \{1, \ldots, c\}$ are the component databases, each exporting n views.

Materialized warehouse schema is given by:

$$\bigcup_{j=1}^{w} f_j\left(\bigcup_{k=1}^{c}\bigcup_{i=1}^{n} v_{ki}\right) + (external\ data)$$

VIEW MANAGEMENT

Local View Manager

The Local View Manager (LVM) is a software module that provides the interface between a component database and the data warehouse. This module is provided for each component database that participates in the federation. LVM implements the specific mapping utility between the local and export schemes, e.g., a function to map hierarchical to relational data model. Generally, there may not be a direct mapping between the local and export schemes, e.g., a table in the export schema can be a view defined by a join between tables. Therefore, a comprehensive catalog is maintained which provides mapping information. Moreover, it makes it easier for the local administrator to maintain the catalog for any change in the local schema.

The other major component of LVM is the generation of differential files for views defined in the export schemes. The implementation of this function depends on the individual component database. For a database that support views, LVM generates a transaction file for each table. At the time of propagation, a differential file is generated for every view. Legacy systems that do not support views will either require modification to the applications such that an entry to a transaction log is made before committing changes, or data can be moved to another database that support views. Alternatively, differential files can be generated periodically by comparing the current and previous snapshots of source tables, but they become inefficient as they grow in size.

LVM has to interact with the Global View Management System (GVMS) for refreshing the data warehouse, and in addition process requests from GVMS for ad-hoc data. We will describe the data structures that help LVM in this process, adopting the assumption that component databases support view-definitions.

View-Manager

The View-Manager maintains information regarding views exported from the local database defined as: *(view-name, view-definition, differential file-name, time created, last refresh-time, next refresh-time, last request-time from GVMS)*, where:

- view-name — uniquely identifies each view that is exported from the local database.
- view-definition — defines the view, select-project-join and attributes of the view.
- differential file-name — uniquely identifies the file to store the net changes that has occurred for the view, between the last and next refresh time.
- time created — a time-stamp indicating when the differential file was created.
- last refresh-time — a time-stamp indicating when the view was last involved in updating the data warehouse.
- next refresh-time — a time-stamp indicating when the view will be next involved in updating the data warehouse.
- last request-time from GVMS — a time-stamp indicating when there was a direct request from GVMS for the view differential file.

Table-Manager

Maintains information regarding tables and the attributes that are taking part in a view defined by: *(table-name, view-name, attribute-name)*, where

- *table-name* — identifies the table taking part in the view definition.
- *view-name* — identifies the view in which the table is used.
- *attribute-name* — identifies attributes of the table that are included in the view definition.

Transaction-Manager

Maintains information regarding the changes on the table between the last and the next refresh times defined by: *(table-name, transaction file-name, trigger-name)*, where:

- *table-name* — identifies the table.
- *transaction file-name* — uniquely identifies the file name that records the transaction for the table, from the last refresh time to the next refresh time.
- *trigger name* — identifies the trigger that will be activated from the table when changes occur on it.

Here we have assumed databases that support triggers and view definitions to give a flavor of the book-keeping functions required. But, for databases that do not support these functions, a different implementation is required. As we are focusing more on framework rather than actual implementation, we do not discuss this issue further.
LVM has to respond to two basic events: 1) Refresh request from GVMS, and 2) ad-hoc request from GVMS. We treat them as separate events, since (1) would require a generation of a differential file for the complete export schema, while (2) requires the generation for a specific view.

Refresh From Local View Manager

At the next refresh cycle, LVM propagates to GVMS a differential file for its export schema. The algorithm (see Figure 2) used for generating these files may vary depending on the type of implementation chosen. Algorithms that generate differential files can be used in systems that support view definitions, while a snapshot differential will have to be used for systems that do not support views. Our model is not constrained by any specific algorithm, as they can be chosen based on the requirements of the data source.

Algorithm Refresh
For (export views) Loop
{Send-Message (site-name, refresh); /* Message to LVM */
$\Delta_i \leftarrow F(g_i(trans\text{-}log(1), trans\text{-}log(2),...,trans\text{-}log(n))$
/* F is View Maintenance algorithm
trans-log(j) is the transaction file-name of table j between refresh intervals.
Δ_i is the differential file-name to be propagated. */
Send-Message(Δ_i) /* Message to GVMS */}
Updated materialized export views at the warehouse:

$$\sum_{i=1}^{m}\sum_{t=1}^{p} v_i(t) + \sum_{i=1}^{m}(v_i + \Delta_i);$$

Materialized warehouse schema after the refresh:

$$\sum_{j=1}^{w} f_j\left(\sum_{i=1}^{m} v_i\right) + \sum_{j=1}^{w} f_j\left(\sum_{i=1}^{m} v_i + \Delta_i\right) + (external\ data)$$

Figure 2. Algorithm Refresh

Request From Global View Management System

Other than periodic refresh, LVM may be requested to send the changes for a specific view defined in its export schema (See Figure 3). This would require the propagation of a differential file for the requested view. We foresee such requests in the earlier stage of the warehouse's existence, since these data will be integrated into the warehouse schema subsequently, and will be updated through the periodic refresh cycles. But we treat these queries as non-preserving. To summarize, LVM provides:

- Mapping between export and local schema of each CDB.
- Generate differential files for the views exported from each CDB.
- Communicate with the Global View Management System (GVMS).
- Maintains the transaction and differential files for each table and view respectively.

Algorithm: Ad-hoc Request

$\forall d_i \notin$ warehouse schema, $\exists\ f_i \mid d_i = f_i\left(\sum_{j=1}^{m} v_j\right)$ /* Transformation */

For (export views in d_i) Loop

{Send-Message (view-reference, site-name); /* Message to LVM */

$\Delta_i \leftarrow F(g_i(\textit{trans-log(1), trans-log(2),.....,trans-log(n))};$

/* F is the view maintenance algorithm */

Send-Message (Δ_i) /* Message to the GVMS */}

Materialized $d_i = f_i\left(\sum_{j=1}^{m}(v_j + \Delta_j)\right)$

Figure 3. Algorithm Ad-hoc Request

Global View Management System (GVMS)

GVMS is a software module that provides similar functionalities to the data warehouse as LVM provides to the component databases. But the GVMS is a global master that coordinates the LVMs and determines the next refresh cycle.

Functionally, GVMS provides:

- A utility to map IDS to the warehouse schema, and also IDS to the export schema of component databases.
- Algorithm to generate differential files for updating the data warehouse schema. It takes as input differential files from LVMs and outputs a differential file corresponding to the warehouse schema.
- Coordination of propagation of differential files from LVMs.
- Requests to LVMs to propagate differential files for ad-hoc data.

The IDWM employs a two stage process for update propagation: in the first stage, a differential file for every export schema is generated; in the second stage, these are combined to generate a differential file for the warehouse. This helps in implementing different algorithms at the warehouse and component databases. We do not see this as a necessity as the model allows for differing implementations, thus providing a framework without making assumptions about the underlying methods.

GVMS consists of various software modules (See Figure 4) that perform different functions. The *Mapper* transforms the warehouse schema to the underlying export views by interacting with IDS. The *Assembler* consolidates results and propagates updates to materialized views, and it also implements the view maintenance algorithm in GVMS. Coordination between these modules is performed by the *Coordinator*, while *Monitor* provides interface between GVMS and LVMs. For ad-hoc queries the coordinator passes the definition to the mapper in order to find relevant export views that fulfill the request. Next, the mapper passes the site addresses of the relevant export views to the monitor to retrieve the differential files. These are buffered, until all requests are obtained by the monitor and passed on to the assembler, which then assembles them according to the query definition and submits the result to the coordinator.

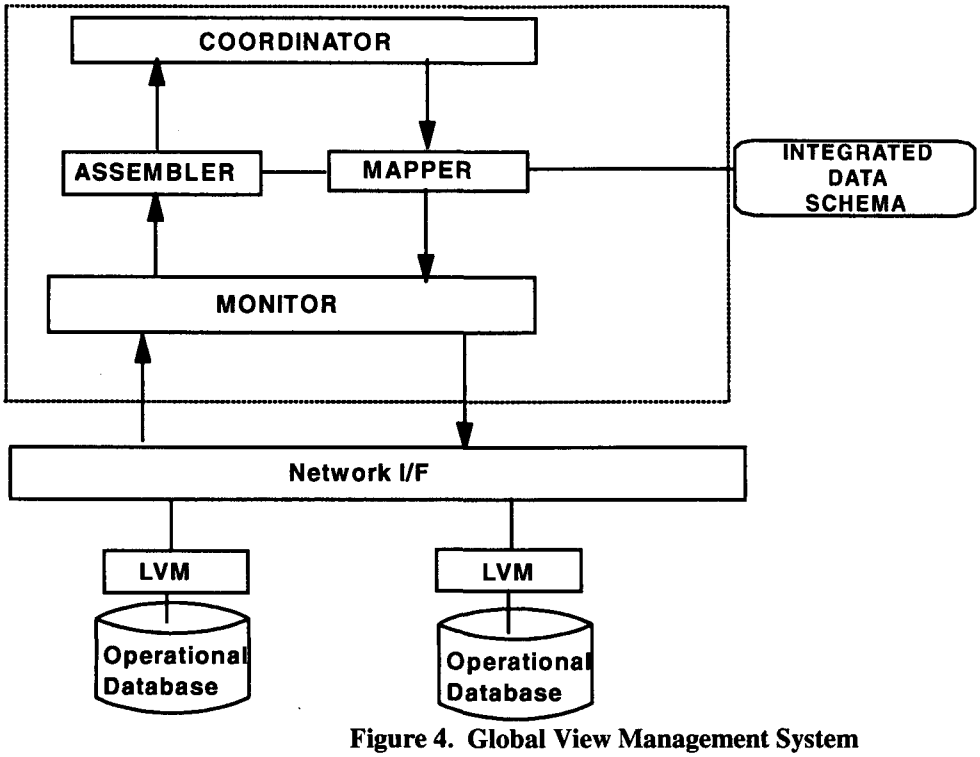

Figure 4. Global View Management System

For periodic update propagation from the component databases, the monitor has an elaborate tracking mechanism through which it ensures that all the differential files are received by the warehouse before the assembler begins propagating. In the event of a network failure, the monitor requests the corresponding local view manager to re-send the differential files.

CONCLUSION AND FUTURE RESEARCH

A great deal of work has been carried out in heterogeneous databases, view maintenance and data warehousing, but only recently have there been some research in integrating them. Our aim is to develop a model and define interfaces required to integrate these methods into a unified architecture. In this process many open and challenging issues that requires further investigation were brought to our focus, such as:

1. View maintenance algorithms for updating aggregates. As the warehouse consists of aggregated data along many dimensions, there are not many generalized algorithms to handle these operations.
2. Algorithms for generating differential files for views that are defined as select-project-join.
3. Handling data changes and update propagation from databases that do not support view definitions.
4. Characterization of data warehouse — present characterization looks at the warehouse as a repository solely for decision-making purposes. What about those applications that require access to enterprise data in a consistent form? Our framework has taken the first step in this direction.
5. We assume the warehouse as an append only database, but what about those that can have updates as well, e.g., to enable error-correction and schema changes.
6. Mechanism to integrate data into the warehouse schema when it is not present when requested for the first time.

We are presently working on defining the various functional modules that have been defined in the IDWM in terms of functions and data structures. Further work is required in defining GVMS and LVM in terms of:

- Functional modules for mapping schemes,
- Functional module to generate differential files, and
- Book-keeping functions.

In addition, we are focusing on developing algorithms to generate differential files on the export and data warehouse schema that are defined as select-project-join. This will be further extended to include aggregates at the data warehouse schema.

To conclude, we have shown that research in the areas of heterogeneous databases and data warehouse have looked at the same problem of data integration quite differently. The former looked at the problem of providing easy access to data residing in different databases, while the latter looked at the problem of bringing the

relevant data to a central repository. We have proposed a data warehouse architecture that is integrated to the data sources, by extending the federated database architecture. Our architecture provides:

- Flexibility — underlying databases can be any legacy system, including those that do not have a view management facility, as long as modules that can supplement the underlying database view facility is included in the LVMs and GVMS.
- Transparent access to operational component databases to enable response to queries needing current data.
- A method of maintaining the data warehouse as a materialized view. Any view management method can be used, as they are pluggable in the modules in LVMs and GVMS.

REFERENCES

Adiba, M. & Lindsay, B.G. (1980) **Database Snapshots**, Proceedings of the Conference on Very Large Databases, pp 86-91.

Agarwal, R. & Dewitt, D.J. (1983) **Updating Hypothetical Databases**, Information Processing Letters, Vol 16, pp 145-146.

Blakely, J.A., Larkson, P. & Tompa, F.W. (1986) **Efficiently Maintaining Views**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 61-71.

Ceri, S. & Widom, J. (1991) **Deriving Production Rules for Incremental View Maintenance**, Proceedings of the Conference on Very Large Databases, pp 577-589.

Gilgor, V.D. & Luckenbaugh, G.L. (1984) **Interconnecting Heterogeneous Database Management Systems**, Computer, Vol 17, 1, pp 33-45.

Griffin, T. & Libkin, L. (1995) **Incremental Maintenance of Views with Duplicates**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 328-339.

Gupta, A. & Mumick, I.S. (1993) **Maintaining Views Incrementally**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 157-166.

Gupta, A., Harinarayan, V. & Quass, D. (1995) **Aggregate-Query Processing in Data Warehousing Environments**, Proceedings of the Conference on Very Large Databases, pp 358-369.

Gupta, A. & Mumick, I.S. (1995) **Maintenance of Materialized Views: Problems, Techniques, and Applications**, Data Engineering Bulletin, Vol 18, 2, pp 3-18.

Hackathorn, R. D. (1995) **Data Warehousing Energizes Your Enterprise**, Datamation, Feb, pp 38-43.

Hammer, J., Gracia-Molina, H., Widom, J., Labio, W. & Zhuge, Y. (1995) **The Stanford Data Warehousing Project**, Data Engineering Bulletin, Vol 18, 2, pp 41-48.

Hanson, E. N. (1987) **A Performance Analysis of View Materialization Strategies**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 440-453.

Inmon, W. H. (1992) **Building the Data Warehouse**, QED.

Inmon, W. H. & Hackathorn, R.D. (1994) **Using the Data Warehouse**, John Wiley.

Kelly, S. (1994) **Data Warehousing: The Route to Mass Customization**, John Wiley.

Lindsay, B., Haas, L., Mohan, C., Pirahesh, H. & Wilms, P. (1986) **A Snapshot Differential Refresh Algorithm**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 53-60.

Roussopoulos, N., Chen, C.M., Kelly, S., Delix, A. & Papakonstantinou, Y. (1995) **The Maryland ADMS Project: Views R Us**, Data Engineering Bulletin, Vol 18, 2, pp 19-28.

Segev, A., Park, J. (1989) **Updating Distributed Materialized Views**, IEEE Transaction on Knowledge and Data Engineering, Vol 2, 1, pp 173-184.

Sheth, A.P. & Larson, J.A. (1990) **Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases**, ACM Computing Surveys, Vol 22, 3, pp 183-236.

Shmueli, O. & Itai, A. (1984) **Maintenance of Views**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 240-255.

Weldon, J. (1995) **Managing Multidimensional Data: Harnessing The Power**, Database Programming and Design, Aug, pp 24-33.

Woodfill, J. & Stonebraker, M. (1983) **An Implementation of Hypothetical Relations**, Proceedings of the Conference on Very Large DataBases, pp 157-165.

Yan, W. & Larson, P. (1995) **Eager Aggregation and Lazy Aggregation**, Proceedings of the Conference on Very Large Databases, pp 345-357.

Zhou, G., Hull, R., King, R. & Franchitti, J. (1995) **Supporting Data Integration and Warehousing Using H2O**, Data Engineering Bulletin, Vol.18, 2, pp 41-48.

Zhuge, Y., Gracia-Molina, H., Hammer, J. & Widom, J. (1995) **View Maintenance in a Warehousing Environment**, Proceedings of the ACM-SIGMOD International Conference on Management of Data, pp 316-327.